NextGen Scientific Publication
American International Journal of Computer Science and Technology

Volume 8 Issue 2, Pg. No. 15-20, AIJCST-V8I2P102, 2026
https://doi.org/10.63282/3117-5481/AIJCST-V8I2P102

Original Article

Securing Microservice Communication across WCF, JAX-RS, and Spring Boot: Authentication, Authorization, and Audit Patterns for Healthcare Interoperability

***Sri Gantikota**
Senior Software Engineer, San Diego, California 92101, USA.

Abstract:

Healthcare integration environments commonly include services written across multiple eras of platform evolution. Windows Communication Foundation services from the early 2010s, Java EE web services using JAX-RS from the mid-2010s, and Spring Boot microservices from the late 2010s and onward all coexist in production at most healthcare technology vendors and many hospital information technology shops. Each stack expresses authentication, authorization, and audit through its own idioms, and securing communication across the heterogeneous environment requires both per-stack rigor and cross-stack consistency. This paper presents authentication, authorization, and audit patterns that work across WCF, JAX-RS, and Spring Boot services, with specific attention to the healthcare integration setting in which these services exchange protected health information. The patterns cover OAuth 2.0 and OpenID Connect token validation in each stack, mutual TLS as a transport-level complement to token-based authentication, attribute-based access control at the service boundary, and audit logging that satisfies both regulatory requirements and operational debugging needs. The paper closes with a discussion of how the patterns interact with API gateway and service mesh deployments, and with the architectural choices that determine whether security can be applied uniformly or has to be reasoned about per stack.

Keywords:

Microservice Security, WCF, JAX-RS, Spring Boot, Oauth 2.0, Openid Connect, JWT, Mtls, API Gateway, Healthcare Interoperability, HIPAA, Authentication, Authorization, Audit Logging.

Article History:

Received: 12.03.2026

Revised: 17.02.2026

Accepted: 26.02.2026

Published: 04.03.2026

1. Introduction

Healthcare integration software accumulates platform diversity over time. A vendor that began shipping integration products in the early 2010s likely has Windows Communication Foundation services in production from that era. The mid 2010s added Java EE web services using JAX-RS, often with WCF clients calling them and JAX-RS clients calling WCF services in the opposite direction. The late 2010s and onward introduced Spring Boot microservices, often in a service mesh or behind an API gateway, that now sit alongside the older services and exchange data with them. Retiring any one of these stacks is rarely feasible in the short term because the services they host are entangled with hospital customer workflows that the vendor cannot easily disrupt.

Securing communication across this heterogeneous environment is a real engineering problem. Each stack expresses authentication, authorization, and audit through its own idioms. WCF has its own configuration model and its own preferred patterns for transport-level and message-level security. JAX-RS has its own filter and interceptor model, with security commonly layered on through frameworks such as Spring Security applied to JAX-RS endpoints or through dedicated JAX-RS security providers. Spring Boot

has Spring Security as the dominant pattern, with OAuth 2.0 client and resource server support built in. A security pattern that is idiomatic in one stack may need substantial translation to be expressed in another.

This paper presents authentication, authorization, and audit patterns that work across WCF, JAX-RS, and Spring Boot services, with specific attention to the healthcare integration setting. The contribution is the cross-stack consistency: the same authentication contract, the same authorization model, and the same audit format expressed in each stack\'s idioms. The cross-stack consistency is what allows a security team to reason about the security properties of the environment as a whole rather than separately per stack.

The rest of the paper is organized as follows. Section 2 describes the three stacks and their security models. Section 3 covers token-based authentication with OAuth 2.0 and OpenID Connect. Section 4 covers mutual TLS as a transport-level complement. Section 5 covers attribute-based authorization. Section 6 covers audit logging. Section 7 covers the role of API gateways and service meshes. Section 8 covers the architectural choices that determine uniformity. Section 9 concludes.

2. The Three Stacks and Their Security Models

2.1. Windows Communication Foundation

Windows Communication Foundation is the .NET framework\'s unified API for building service-oriented applications. WCF supports multiple bindings that encapsulate the choice of transport, encoding, and security. The bindings include basicHttpBinding, wsHttpBinding, netTcpBinding, and others. Security in WCF is configured through binding configuration that selects transport-level security, message-level security, or both, and through behavior configuration that selects the credential type and the authorization model. The WCF security model is mature and capable but its configuration surface is large and the right choice for a given service depends on the deployment context.

2.2. Java API for RESTful Web Services

Java API for RESTful Web Services, JAX-RS, is the Java EE specification for building REST APIs. JAX-RS itself is intentionally minimal on security; security is layered on through providers such as the security context API and through container or framework-level integration. In practice, a JAX-RS application running in a Java EE container uses the container\'s security model. A JAX-RS application running outside a container, perhaps in an embedded server, uses framework-level security such as Spring Security or Apache Shiro. The provider model is flexible but creates the situation in which the security of two JAX-RS services in the same product can be configured very differently.

2.3. Spring Boot

Spring Boot is the convention-over-configuration framework for building Spring-based applications. Spring Security is the dominant security framework in Spring Boot applications and provides comprehensive support for authentication, authorization, and integration with OAuth 2.0 and OpenID Connect. Spring Security\'s configuration model is more uniform than the WCF and JAX-RS alternatives, in the sense that the same Spring Security idioms apply across all Spring Boot services, but the breadth of Spring Security\'s capabilities means that learning what each idiom does is still an investment for new developers.

3. Token-Based Authentication

3.1. OAuth 2.0 and OpenID Connect

OAuth 2.0 is the dominant standard for delegated authorization. OpenID Connect adds an identity layer on top of OAuth 2.0 for federated authentication. Both standards are well-established and well-supported across the ecosystem. The patterns presented here use OAuth 2.0 for service-to-service authorization and OpenID Connect for user authentication, with JSON Web Tokens as the token format. The choice is the same across the three stacks; only the implementation in each stack differs.

3.2. JWT Validation in WCF

WCF validates JWTs through a custom service authorization manager that examines the incoming request, extracts the bearer token from the Authorization header, validates the token\'s signature and claims, and populates the service\'s security context. The implementation requires writing the authorization manager and configuring the service to use it, but the underlying validation logic is identical to the logic the other stacks would use. The token\'s claims become the basis for the authorization decisions described in Section 5.

3.3. JWT Validation in JAX-RS

JAX-RS validates JWTs through a request filter that runs before resource methods. The filter extracts the bearer token, validates it, and populates the security context. The same filter pattern is used regardless of which underlying framework provides the security primitives. The configuration of the filter is per service, which means that consistency across JAX-RS services requires that the same filter implementation be packaged into each service rather than relying on each service to configure its own.

3.4. JWT Validation in Spring Boot

Spring Boot validates JWTs through Spring Security\'s OAuth 2.0 Resource Server support. The configuration is a small block that declares the issuer URI and the JWK set URI, after which Spring Security handles the validation transparently. The simplicity of the Spring Boot configuration is in marked contrast to the WCF and JAX-RS alternatives, which require more explicit code. The simplicity is a benefit, but it also means that a developer who has only worked in Spring Boot may underestimate the work required to achieve the same outcome in the other stacks.

3.5. Token Issuance

Token issuance is centralized in an identity provider that all three stacks trust. Common choices include Keycloak as an open-source identity provider, Autho as a hosted alternative, and the identity providers built into the major cloud platforms. The choice of identity provider is independent of the stack mix. The relevant operational property is that the identity provider\'s public keys are reachable from all three stacks for token validation, which in a healthcare integration setting typically means the identity provider sits on a network reachable from the integration tier.

4. Mutual TLS as a Transport-Level Complement

4.1. Why mTLS in Addition to Tokens

Mutual TLS authenticates both the client and the server at the TLS layer through X.509 certificates. mTLS does not replace token-based authentication; it complements it. Token-based authentication identifies the user or service at the application layer and carries authorization claims. mTLS identifies the calling service at the transport layer and provides a strong identity anchor that the application-layer token can be tied to. The combination defends against both compromised tokens, which the mTLS layer would still require a valid certificate to use, and compromised certificates, which the token layer would still require a valid token.

4.2. mTLS in Each Stack

WCF supports mTLS through binding configuration that selects transport security with certificate authentication. JAX-RS supports mTLS through the underlying servlet container or embedded server, which is configured at the transport layer rather than in the JAX-RS application code. Spring Boot supports mTLS through the embedded server, similarly configured at the transport layer. The configuration surface is different in each stack but the operational outcome is the same: a connection cannot be established without a valid client certificate.

4.3. Certificate Lifecycle

Certificate lifecycle management is the operational cost of mTLS. Certificates expire and must be rotated, certificate authorities must be trusted across all the stacks involved, and revocation must work in a timely way when a certificate is compromised. The healthcare integration setting typically uses a private certificate authority operated by the vendor or by an enterprise customer. The lifecycle is managed through tooling that automates issuance and rotation; manual certificate management at scale is the source of most mTLS operational failures in practice.

5. Attribute-Based Authorization

5.1. Why Attribute-Based

Role-based access control assigns permissions to roles and roles to users. Attribute-based access control assigns permissions based on attributes of the user, the resource, the action, and the environment. Healthcare authorization typically requires attribute-based reasoning because the permission to access a record may depend on attributes such as the requesting clinician\'s relationship to the patient, the time of day, the kind of record, and the explicit consent the patient has given. Pure role-based access control does not express these conditions cleanly.

5.2. Authorization at the Service Boundary

Authorization is evaluated at the service boundary, where the incoming request meets the resource. The evaluation considers the claims in the validated token, the attributes of the resource being accessed, the attributes of the action being attempted, and the environmental context. The result is either permit or deny. The decision logic is encapsulated in a policy engine, which can be embedded in the service or hosted as a separate authorization service. The Open Policy Agent project is a representative example of a policy engine that can be invoked from any of the three stacks.

5.3. Policy Externalization

Externalizing policies from service code lets the policies be updated without redeploying the service, and it lets the same policy be enforced consistently across services. The pattern is to write policies in a declarative language that the policy engine interprets, to deploy the policies as artifacts separate from the service binaries, and to invoke the policy engine for each authorization decision. The trade-off is that the service must perform a policy engine call on each request, which has performance implications that the architecture has to account for.

5.4. Caching Authorization Decisions

Caching authorization decisions can reduce the performance impact of policy engine calls. The cache key must include all the inputs that the policy considered, so that a stale cache does not return an incorrect decision when one of the inputs has changed. The cache must also be invalidated when a policy changes. The practical pattern in a healthcare setting is a short-lived per-process cache that captures the decisions made in the recent past, sized to absorb the burst access patterns the application produces while keeping the staleness bounded.

6. Audit Logging

6.1. Audit Requirements

Audit logging in a healthcare setting must satisfy regulatory requirements that go beyond operational logging. The HIPAA Security Rule requires audit controls that record and examine activity in systems containing protected health information. The audit log must capture who did what to which record at what time, and it must be tamper-resistant so that it can serve as evidence in an investigation. The audit log is therefore a first-class concern in service design, not an afterthought.

6.2. Audit Format

A consistent audit format across the three stacks lets the security team analyze audit data without per-stack parsing. The format includes the time of the event, the identity of the principal that initiated the event, the action attempted, the resource affected, the outcome, and any relevant context such as the source network address. The Integrating the Healthcare Enterprise community has published audit format specifications based on the IETF Common Event Format and the Audit Trail and Node Authentication profile that healthcare vendors commonly use as a reference.

6.3. Audit in Each Stack

Each stack emits audit events through its own logging facilities, but the events are routed to a common audit log store. WCF emits audit events from the service authorization manager and from custom interceptors. JAX-RS emits audit events from request filters. Spring Boot emits audit events from Spring Security listeners and from custom interceptors. The common store receives events in the consistent format described in Section 6.2 regardless of the originating stack.

6.4. Audit Log Integrity

Audit log integrity is achieved through append-only storage and tamper-evidence mechanisms such as cryptographic chaining or signed batches. The detailed mechanism depends on the operational environment, but the requirement is consistent: an attacker who has compromised a service must not be able to alter the audit log to conceal their activity. The audit log store is therefore protected differently from the production data stores, often with separate access credentials and stricter monitoring.

7. API Gateways and Service Meshes

7.1. Centralizing Concerns at the Gateway

An API gateway centralizes cross-cutting concerns including authentication, rate limiting, request logging, and protocol translation. In an environment with the stack diversity described here, the API gateway is the place where the per-stack security idioms can be unified into a single edge experience. The gateway validates tokens, applies rate limits, and forwards the request to the appropriate backend service with the validated identity propagated in headers the backend can trust.

7.2. Service Mesh for East-West Traffic

A service mesh handles east-west traffic between services in the same way the API gateway handles north-south traffic from clients. The mesh provides mTLS between services without requiring each service to configure it, and it can perform authentication and authorization at the sidecar level. The pattern is useful in environments with many services and substantial inter-service communication; it is less useful in environments with a small number of services where the configuration cost of the mesh outweighs its benefits.

7.3. Defense in Depth

Centralizing concerns at the gateway or in the mesh does not remove the need for in-service security. A compromised gateway should not give an attacker the run of the backend services. The pattern is defense in depth: the gateway enforces the policy at the edge, and each service enforces the policy independently. Duplicated enforcement is the cost; resistance to gateway compromise is the benefit.

8. Architectural Choices and Uniformity

8.1. What Can Be Made Uniform

The authentication contract, the token format, the authorization policy language, and the audit format can all be made uniform across the three stacks. Uniformity at this layer is what allows the security team to reason about the environment as a whole. Stack-specific implementation work is required to achieve the uniformity, but the work is bounded and pays back through the reduced cognitive load of reasoning about a uniform system.

8.2. What Cannot Be Made Uniform

Per-stack idioms cannot be made uniform without retiring the stacks. The WCF configuration model is what it is. The JAX-RS provider model is what it is. The Spring Security configuration model is what it is. The practical approach is to accept the per-stack idioms at the implementation layer and to maintain uniformity at the contract layer. The cost of this approach is documentation that explains the contract in each stack\'s terms; the benefit is the avoidance of attempts to harmonize stack-internal mechanisms that cannot actually be harmonized.

8.3. Migration Pressure

The stack diversity creates ongoing migration pressure as older stacks become harder to maintain. The pattern in most organizations is to migrate services incrementally, retiring WCF services as their hosts are retired and consolidating onto Spring Boot or onto whatever has succeeded it. The migration is years-long work that the security patterns must accommodate without disruption. The contract-layer uniformity described in Section 8.1 is what makes the migration tractable, because new services can be written against the same contract without introducing security-relevant differences.

9. Conclusion

Securing microservice communication in a healthcare integration environment with WCF, JAX-RS, and Spring Boot services in production requires attention to both per-stack rigor and cross-stack consistency. Each stack expresses authentication, authorization, and audit through its own idioms, and the security patterns must be expressed in each idiom while preserving a uniform contract at the boundary. OAuth 2.0 and OpenID Connect with JWT tokens, mutual TLS as a transport-level complement, attribute-based authorization with policies externalized from service code, and a consistent audit format across all stacks together produce a security posture that satisfies both the regulatory requirements of healthcare and the operational requirements of a mature integration environment. API gateways and service meshes centralize cross-cutting concerns at the edge and between services without removing the need for defense in depth at each service. The contract-layer uniformity that the patterns produce is what allows the security team to reason about the environment as a whole and what makes the inevitable stack migrations tractable over time.

Acknowledgments

This work draws on engineering practice in healthcare integration across multiple roles at Merge Healthcare, IBM, and the University of California, San Diego. The author thanks colleagues across these organizations whose practical experience informed the patterns presented.

Conflicts of Interest

The author declares that there is no conflict of interest concerning the publishing of this paper.

References

[1] Hardt, D. The OAuth 2.0 Authorization Framework. IETF RFC 6749, October 2012. https://scholar.google.com/scholar?hl=en&q=The OAuth 2.0 Authorization Framework

[2] Jones, M., Bradley, J., and Sakimura, N. JSON Web Token (JWT). IETF RFC 7519, May 2015. https://scholar.google.com/scholar?hl=en&q=JSON Web Token (JWT)

[3] Sakimura, N. et al. OpenID Connect Core 1.0. OpenID Foundation, November 2014. https://scholar.google.com/scholar?hl=en&q=et al

[4] Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 8446, August 2018. https://scholar.google.com/scholar?hl=en&q=The Transport Layer Security (TLS) Protocol Version 1.3

[5] National Institute of Standards and Technology. Guide to Attribute-Based Access Control Definition and Considerations, NIST Special Publication 800-162. https://scholar.google.com/scholar?hl=en&q=Guide to Attribute-Based Access Control Definition and Considerations, NIST Special Publication 800-162

[6] Open Policy Agent project. OPA documentation. https://www.openpolicyagent.org/docs/ | https://scholar.google.com/scholar?hl=en&q=OPA documentation

[7] Microsoft Corporation. Windows Communication Foundation documentation. https://scholar.google.com/scholar?hl=en&q=Windows Communication Foundation documentation

[8] Eclipse Foundation. Jakarta RESTful Web Services Specification. https://scholar.google.com/scholar?hl=en&q=Jakarta RESTful Web Services Specification

[9] Spring Project. Spring Security reference documentation. https://scholar.google.com/scholar?hl=en&q=Spring Security reference documentation

[10] Spring Project. Spring Boot reference documentation. https://scholar.google.com/scholar?hl=en&q=Spring Boot reference documentation

[11] Red Hat. Keycloak documentation. https://www.keycloak.org/documentation |https://scholar.google.com/scholar?hl=en&q=Keycloak documentation

[12] Cloud Native Computing Foundation. Istio service mesh documentation. https://istio.io/latest/docs/ | https://scholar.google.com/scholar?hl=en&q=Istio service mesh documentation

[13] Cloud Native Computing Foundation. Envoy proxy documentation. https://www.envoyproxy.io/docs/ | https://scholar.google.com/scholar?hl=en&q=Envoy proxy documentation

[14] Integrating the Healthcare Enterprise. IHE IT Infrastructure Technical Framework, Audit Trail and Node Authentication Profile. https://scholar.google.com/scholar?hl=en&q=IHE IT Infrastructure Technical Framework, Audit Trail and Node Authentication Profile

[15] United States Department of Health and Human Services. Health Insurance Portability and Accountability Act Security Rule, 45 CFR Part 164 Subpart C. https://scholar.google.com/scholar?hl=en&q=Health Insurance Portability and Accountability Act Security Rule, 45 CFR Part 164 Subpart C

[16] Open Web Application Security Project. OWASP API Security Top Ten, 2023 edition. https://scholar.google.com/scholar?hl=en&q=OWASP API Security Top Ten, 2023 edition

[17] National Institute of Standards and Technology. Security Strategies for Microservices-based Application Systems, NIST Special Publication 800-204. https://scholar.google.com/scholar?hl=en&q=Security Strategies for Microservices-based Application Systems, NIST Special Publication 800-204

[18] Newman, S. Building Microservices, Second Edition. O\'Reilly Media, 2021. https://scholar.google.com/scholar?hl=en&q=Building Microservices, Second Edition

[19] Richardson, C. Microservices Patterns. Manning Publications, 2018. https://scholar.google.com/scholar?hl=en&q=Microservices Patterns